Multi Thread Programming in Java

- Resources
  - API docs
    - http://download.oracle.com/javase/6/docs/api/
      - java.lang.Thread, java.lang.Runnable
      - java.lang.Object, java.util.concurrent
  - Tutorials
    - http://download.oracle.com/javase/tutorial/essential/concurrency/index.html
    - http://download.oracle.com/javase/tutorial/essential/concurrency/procthread.html
  - Introduction to Java Threads
    - http://www.javaworld.com/javaworld/jw-04-1996/jw-04-threads.html
  - Thread safety
    - http://en.wikipedia.org/wiki/Thread-safety
    - http://www.javaworld.com/jw-08-1998/jw-08-techniques.html

Coverage

- Thread class
  - run, start methods
  - yield, join
  - sleep
- Synchronization
  - synchronized methods & objects
  - wait/notify/notifyAll
  - conditions

java.lang.Thread

- Two techniques to create threads in java
- 1) implementing the Runnable interface
  - The Runnable interface should be ***implemented*** by any class whose instances are intended to be executed by a thread. The class must define a method, called *run*, with no arguments.
  - invoke Thread constructor with an instance of this Runnable class
  - See pages 162 and 164 in text for an example
- 2) extending Thread
  - Define a subclass of java.lang.Thread
    - Define a *run* method
  - In another thread (e.g., the main), create an instance of the Thread subclass
    - Then, call *start* method of that instance

3

Example 1

- Create 2 threads from the Main, then start them
- Threads will be instances of different thread sub-classes

```java
class MyThreadA extends Thread {
    public void run() { // entry point for thread
            for (;;) {
                    System.out.println("hello world1");
            }
    }
}

class MyThreadB extends Thread {
    public void run() { // entry point for thread
            for (;;) {
                    System.out.println("hello world2");
            }
    }
}

public class Main1 {
    public static void main(String [] args) {
            MyThreadA t1 = new MyThreadA();
            MyThreadB t2 = new MyThreadB();
            t1.start();
            t2.start();
            // main terminates, but in Java the other threads keep running
            // and hence Java program continues running
    }
}
```

hello world2
hello world2
hello world1
hello world2
hello world1
hello world2
hello world2
hello world1
hello world1
hello world1
hello world1
hello world2
hello world1
hello world1
hello world2
hello world2
hello world1
hello world1
hello world2
hello world2
hello world1
hello world1
hello world2

Example 2

- Create 2 threads from the Main, then start them
- Threads will be instances of the same thread sub-class
- Use argument of constructor of new thread class to pass text name of thread, e.g., "thread1" and "thread2"
  - Data member provides different data per thread (i.e., then name)
  - A data member can also be used to share data

7

```java
class MyThread extends Thread {
        private String name;

        public MyThread(String name) {
                this.name = name;
        }

        public void run() {
                for (;;) {
                        System.out.println(name + ": hello world");
                }
        }
}

public class Main2 {
        public static void main(String [] args) {
                MyThread t1 = new MyThread("thread1");
                MyThread t2 = new MyThread("thread2");
                t1.start(); t2.start();
        }
}
```

thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread2: hello world

See the variation in output: This variation in output is called a "race condition" (often race conditions are bugs in programs)

java.lang.Thread

- public static void yield();
  - Method of java.lang.Thread
  - Thread gives up CPU for other threads ready to run

```java
class MyThread extends Thread {
        private String name;

        public MyThread(String name) {
                this.name = name;
        }

        public void run() {
                for (;;) {
                        System.out.println(name + ": hello world");
                        yield();
                }
        }
}

public class Main3 {
        public static void main(String [] args) {
                MyThread t1 = new MyThread("thread1");
                MyThread t2 = new MyThread("thread2");
                t1.start(); t2.start();
        }
}
```

Some Output

thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world

Notice the alternation
of output

More Thread Members: *join*

- public final void join();

  MyThread t1 = new MyThread("thread1");
  t1.start();
  t1.join();

  – Wait until the thread is "not alive"
  – Threads that have completed are "not alive" as are threads that have not yet been started

- public static void sleep (long millis) throws InterruptedException;
  – Makes the currently running thread sleep (block) for a period of time
  – The thread does not lose ownership of any monitors.
  – InterruptedException - if another thread has interrupted the current thread.

Join Example

```java
class MyThread extends Thread {
    public void run() {
        for (int i=0; i < 1000; i++) {
            System.out.println("hello world1");
        }
    }
}

public class Main4 {
    public static void main(String [] args) {
        MyThread t1 = new MyThread();
        t1.start();
    try {
        t1.join(); // wait for the thread to terminate
    } catch (InterruptedException e) {
        System.out.println("ERROR: Thread was interrupted");
    }

    System.out.println("Thread is done!");
    }
}
```

Some output

…
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
Thread is done!

Thread State

- public Thread.State getState()
  - Returns the state of this thread. This method is designed for use in monitoring of the system state, not for synchronization control

```
public static enum Thread.State
extends Enum<Thread.State>
```

A thread state. A thread can be in one of the following states:

- **NEW**
  A thread that has not yet started is in this state.
- **RUNNABLE**
  A thread executing in the Java virtual machine is in this state.
- **BLOCKED**
  A thread that is blocked waiting for a monitor lock is in this state.
- **WAITING**
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- **TIMED_WAITING**
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- **TERMINATED**
  A thread that has exited is in this state.

A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.

http://java.sun.com/javase/6/docs/api/java/lang/Thread.State.html

Thread Scheduling in Java

- public final void setPriority(int newPriority);
- public final int getPriority();
- public static final int MAX_PRIORITY

  // on my system: 10; Mac OS X 2/21/05

- public static final int MIN_PRIORITY

  // on my system: 1; Mac OS X 2/21/05

- Scheduling
  - Priority inherited from parent, but can be changed
  - Higher priority threads generally run before lower priority threads
  - For equal priority threads, best to call yield() intermittently to handle JVM's with user-level threading (i.e., no time-slicing)

Sharing Data Across Java Threads

- Consider the situation where a parent thread wants to pass data to a child thread

  – e.g., so that child can change data and parent can have access to the changed data

- How can this be done?

- Can pass an object instance to the child thread constructor, and retain that object instance in a data member

```java
class SharedData {
        public int a = 0;
        public String s = null;

        public SharedData() {
                a = 10;
                s = "Test";
        }
}

class MyThread extends Thread {
        private SharedData m_data = null;

        public MyThread(SharedData data) {
                m_data = data;
        }

        public void run() {
                for (;;) {
                        m_data.a++;
                }
        }
}
```

```
public class Main5 {
        public static void main(String [] args) {
                SharedData data = new SharedData();
                MyThread t1 = new MyThread(data);
                t1.start();

                for (;;) {
                        data.a--;
                }
        }
}
```

If we have multiple threads accessing this shared data, how do we synchronize access to ensure it remains in a consistent state?

Basic Tools for Synchronization

- Synchronized methods
- Synchronized objects
- Methods
  - wait
  - notify
  - notifyAll
- Also should talk about condition variables in Java

Synchronized Methods: Monitors

- synchronized keyword used with a method
  - E.g.,

    public synchronized void SetValue() {

       // Update instance data structure.

       // When the thread executes here, it exclusively has the ***monitor lock***

    }

  - Provides *instance-based* mutual exclusion
    - A lock is implicitly provided-- allows at most one thread to be executing the method at one time
  - Used on a per method basis; not all methods in a class have to have this
    - But, you'll need to design it right!!

Difference: Synchronized vs. Non-synchronized

- Class with synchronized methods

  - How many threads can access the methods of an object?

- Class with no synchronized methods

  - How many threads can access the methods of an object?

Example

- Construct a queue (FIFO) data structure that can be used by two threads to access the queue data in a synchronized manner
  - Producer thread: Adds data into queue
  - Consumer thread: Removes data from queu
- For one instance of the queue, only one thread should be able to modify the queue, i.e., we should have mutual exclusion on methods of one instance of the queue

```java
// Generic synchronized queue. This can be instantiated with any Object type.
// Only 1 thread can use Add or Remove at a time
class SynchQueue<DataType> {
    public LinkedList<DataType> l;

    SynchQueue () {
        l = new LinkedList<DataType>();
    }

    public synchronized void Add(DataType elem) { // add an element to queue
        l.addLast(elem);
    }

    public synchronized DataType Remove() { // remove an element from queue
        if (l.size() > 0) {
            return l.removeFirst();
        } else {
            return null;
        }
    }
}
```

```java
class Producer extends Thread {
    SynchQueue<Integer> q;
    int curr;

    Producer (SynchQueue<Integer> q) {
        this.q = q;
        curr = 1;
    }

    public void run() {
        for (;;) {
            Integer i = new Integer(curr);
            q.Add(i);
            curr++;
        }
    }
}
```

```java
class Consumer extends Thread {
    SynchQueue<Integer> q;

    Consumer (SynchQueue<Integer> q) {
        this.q = q;
    }

    public void run() {
        for (;;) {
            Integer i = q.Remove();
            if (i != null) {
                System.out.print(i + " ");
            }
        }
    }
}
```

```java
import java.util.LinkedList; // not synchronized

public class SynchMainGeneric {
    public static void main(String args[]) {
        SynchQueue<Integer> q = new SynchQueue<Integer>();
        Producer p = new Producer(q);
        Consumer c = new Consumer(q);
        p.start();
        c.start();
    }
}
```

26

Let's run this and see what happens

Terminal — bash — ttys002

```
nomad203-115:Desktop chris$ javac SynchMain.java
nomad203-115:Desktop chris$ java SynchMain
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 5
7 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107
108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147
148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167
168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187
188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227
228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247
248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267
268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287
288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307
308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327
328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347
348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367
368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387
388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407
408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427
428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447
448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467
```

27

```
26 5327 5328 5329 5330 5331 5332 5333 5334 5335 5336 5337 5338 5339 5340 5341 53
42 5343 5344 5345 5346 5347 5348 5349 5350 5351 5352 5353 5354 5355 5356 5357 53
58 5359 5360 5361 5362 5363 5364 5365 5366 5367 5368 5369 5370 5371 5372 5373 53
74 5375 5376 5377 5378 5379 5380 5381 5382 5383 5384 5385 5386 5387 5388 5389 53
90 5391 5392 5393 5394 5395 5396 5397 5398 5399 5400 5401 5402 5403 5404 5405 54
06 5407 5408 5409 5410 5411 5412 5413 5414 5415 5416 5417 5418 5419 5420 5421 54
22 5423 5424 5425 5426 5427 5428 5429 5430 5431 5432 5433 5434 5435 5436 5437 54
38 5439 5440 5441 5442 5443 5444 5445 5446 5447 5448 5449 5450 5451 5452 5453 54
54 5455 5456 5457 5458 5459 5460 5461 5462 5463 5464 5465 5466 5467 5468 5469 54
70 5471 5472 5473 5474 5475 5476 5477 5478 5479 5480 5481 5482 5483 5484 5485 54
86 5487 5488 5489 5490 5491 5492 5493 5494 5495 5496 5497 5498 5499 5500 5501 55
02 5503 5504 5505 5506 5507 5508 5509 5510 5511 5512 5513 5514 5515 5516 5517 55
18 5519 5520 5521 5522 5523 5524 5525 5526 5527 5528 5529 5530 5531 5532 5533 55
34 5535 5536 5537 5538 5539 5540 5541 5542 5543 5544 5545 5546 5547 5548 5549 55
50 5551 5552 5553 5554 5555 5556 5557 5558 5559 5560 5561 5562 5563 5564 5565 55
66 5567 5568 5569 5570 5571 5572 5573 5574 5575 5576 5577 5578 5579 5580 5581 55
82 5583 5584 5585 5586 5587 5588 5589 5590 5591 5592 5593 5594 5595 5596 5597 55
98 5599 5600 5601 5602 5603 5604 5605 5606 5607 5608 5609 5610 5611 5612 5613 56
14 5615 5616 5617 5618 5619 5620 5621 5622 5623 5624 5625 5626 5627 5628 5629 56
30 5631 5632 5633 5634 5635 5636 5637 5638 5639 5640 5641 5642 5643 5644 5645 56
46 5647 5648 5649 5650 5651 5652 5653 5654 5655 5656 5657 5658 5659 5660 5661 56
62 5663 5664 5665 5666 5667 5668 5669 5670 5671 5672 5673 5674 5675 5676 5677 56
78 5679 5680 5681 5682 5683 5684 5685 5686 5687 5688 5689 5690 5691 5692 5693 56
94 5695 5696 5697 5698 5699 5700 5701 5702 5703 5704 5705 5706 5707 5708 5709 57
```

28

Ooops! What happened?

```
14 9615 9616 9617 9618 9619 9620 9621 9622 9623 9624 9625 9626 9627 9628 9629 96
30 9631 9632 9633 9634 9635 9636 9637 9638 9639 9640 9641 9642 9643 9644 9645 96
46 9647 9648 9649 9650 9651 9652 9653 9654 9655 9656 9657 9658 9659 9660 9661 96
62 9663 9664 9665 9666 9667 9668 9669 9670 9671 9672 9673 9674 9675 9676 9677 96
78 9679 9680 9681 9682 9683 9684 9685 9686 9687 9688 9689 9690 9691 9692 9693 96
94 9695 9696 9697 9698 9699 9700 9701 9702 9703 9704 9705 9706 9707 9708 9709 97
10 9711 9712 9713 9714 9715 9716 9717 9718 9719 9720 9721 9722 9723 9724 9725 97
26 9727 9728 9729 9730 9731 9732 9733 9734 9735 9736 9737 9738 9739 9740 9741 97
42 9743 9744 9745 9746 9747 9748 9749 9750 9751 9752 9753 9754 9755 9756 9757 97
58 9759 9760 9761 9762 9763 9764 9765 9766 9767 9768 9769 9770 9771 9772 9773 97
74 9775 9776 9777 9778 9779 9780 9781 9782 9783 9784 9785 9786 9787 9788 9789 97
90 9791 9792 9793 9794 9795 9796 9797 9798 9799 9800 9801 9802 9803 9804 9805 98
06 9807 9808 9809 9810 9811 9812 9813 9814 9815 9816 9817 9818 9819 9820 9821 98
22 9823 9824 9825 9826 9827 9828 9829 9830 9831 9832 9833 9834 9835 9836 9837 98
38 9839 9840 9841 9842 9843 9844 9845 9846 9847 9848 9849 9850 9851 9852 9853 98
54 9855 9856 9857 9858 9859 9860 9861 9862 9863 9864 9865 9866 9867 9868 9869 98
70 9871 9872 9873 9874 9875 9876 9877 9878 9879 9880 9881 9882 9883 9884 9885 98
86 9887 9888 9889 9890 9891 9892 9893 9894 9895 9896 9897 9898 9899 9900 9901 99
02 9903 9904 9905 9906 9907 9908 9909 9910 9911 9912 9913 9914 9915 9916 Excepti
on in thread "Thread-0" java.lang.OutOfMemoryError: Java heap space
        at java.util.LinkedList.addBefore(LinkedList.java:778)
        at java.util.LinkedList.addLast(LinkedList.java:164)
        at SynchQueue.Add(SynchMain.java:23)
        at Producer.run(SynchMain.java:47)
```

- This implementation has a problem! The Consumer prints, which slows it down a LOT, and thus the producer is faster, and thus the producer fills up the queue, and causes heap space to run out!!

- This is a kind of *race condition*
  - The results depend on the speed of execution of the two processes

- Would like to alter this program to limit the maximum number of items that are stored in the queue.

- *Goal*: have the producer *block* (wait) when the queue reaches some fixed size limit

Also

- Better to have the Remove block (wait) when the queue is empty

- I.e., presently we are doing a "busy wait" (also called polling)

- We are repeatedly checking the queue to see if it has data, and using up too much CPU time doing this

wait method (see also java.lang.Object)

- Does a blocking (not busy) wait
- Relative to an Object
  - E.g., Used within a synchronized method
- Releases lock on Object and waits until a condition is true
  - Blocks calling process until notify() or notifyAll() is called on same object instance (or exception occurs)
- Typically used within a loop to re-check a condition
- wait(long millis); // bounded wait

notify and notifyAll methods (see also java.lang.Object)

- Stop a process from waiting– wakes it up
- Relative to an Object
  - E.g., Used within a synchronized method
- Wakes up a blocked thread (notify) or all blocked threads (notifyAll)
  - One woken thread reacquires lock; The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.
- For notify, if more than one thread available to be woken, then one is picked

Typical use of *wait* within a synchronized method

```
while (condition not true) {
    try {
        wait(); // this.wait();
    } catch {
        System.out.println("Interrupted!");
    }
}
// After loop, condition now true & thread
// has monitor lock for this object instance
```

Example

- Extend the example from before:

  – a queue (FIFO) data structure that can be used by two threads to access the queue data in a synchronized manner

- This time, use wait & notify to block the Producer thread if the queue is full, and block Consumer thread if the queue is empty

Re-checking Monitor Conditions

- wait/notify
  - After receiving a notify, a process waiting on a condition may not be next to gain access to monitor (to the data)
    - E.g., occurs if notifyAll used
  - Process may need to re-check the conditions upon which it was waiting
- An "awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object." (http://java.sun.com/j2se/1.5.0/docs/api/)

InterruptedException

- Wait can be woken by the exception, I.e., for reasons other than notify

- Sometimes this can be handled as part of the process of re-checking conditions

- There is another way to handle it too

Exception in Wait

```
// In a synchronized method

// check your condition, e.g., with a semaphore
// operation, test "value" member variable

if /* or while */ (/* condition */) {
    boolean interrupted;
    do {
        interrupted = false;
        try {
            wait();
        } catch (InterruptedException e) {
            interrupted = true;
        }
    } while (interrupted);
}
```

Only allows release
from wait caused by
notify or notifyAll

Synchronized Blocks

- Synchronized methods
  - Implicitly lock is on *this* object
- Synchronized *blocks*
  - lock on an arbitrary, specified object
  - similar to condition variables in monitors
  - but need to have a synchronized block around an object before wait/notify used
  - use wait/notify on the object itself

Syntax

synchronized (object) {

    // object.wait()

    // object.notify()

    // object.notifyAll()

}

- For example, this allows you to synchronize just a few lines of code, or to synchronize on the basis of an arbitrary object

Another Example

- Suppose in a Global File Table, suppose that per open file you keep an

Object Lock;

- you can then use a synchronized block to make sure that some operations only get done in a mutually exclusive manner on the file

```
synchronized (file[i].Lock) {
        // if we get to here we're the only one
        // accessing file i
}
```

Conditions

- Java interface:
  - http://download.oracle.com/javase/6/docs/api/java/util/concurrent/locks/Condition.html

- Let's you have multiple independent wait events for a monitor

- So, have one monitor lock, but can wait within the monitor for more than one reason

- Use await/signal (not wait/notify)

- Also: Must have explicit lock (don't use *synchronized* keyword)

- Lock monitor as ***very first*** thing you do

- Unlock monitor as ***very last*** thing you do

END!